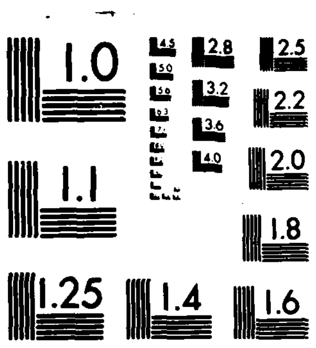
AD-A129 153   THE INTELLIGENT PROGRAM EDITOR: A KNOWLEDGE BASED SYSTEM FOR SUPPORTING P..(U) ADVANCED INFORMATION AND DECISION SYSTEMS MOUNTAIN VIEW CA   D G SHAPIOR ET AL.   1/1

UNCLASSIFIED   MAR 83 AFOSR-TR-83-0488 F49620-81-C-0067   F/G 9/2   NL

END
DATE
FILMED
7-83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

2

AD A129153

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 83-0488	2. GOVT ACCESSION NO. AD-A129153	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE INTELLIGENT PROGRAM EDITOR: A KNOWLEDGE BASED SYSTEM FOR SUPPORTING PROGRAM AND DOCUMENTATION MAINTENANCE		5. TYPE OF REPORT & PERIOD COVERED TECHNICAL
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Daniel G. Shapior and Brian P. McCune		8. CONTRACT OR GRANT NUMBER(s) F49620-81-C-0067
9. PERFORMING ORGANIZATION NAME AND ADDRESS Advanced Information and Decision Systems 201 San Antonio Circle, Suite 286, Mountain View CA 94040		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Mathematical & Information Sciences Directorate Air Force Office of Scientific Research Bolling AFB DC 20332		12. REPORT DATE MAR 83
		13. NUMBER OF PAGES 7
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper presents work in progress towards a program development and maintenance aid called the Intelligent Program Editor (IPE), which applies artificial intelligence techniques to the task of manipulating and analyzing programs. The IPE is a knowledge based tool: it gains its power by explicitly representing textual, syntactic, and many of the semantic (meaning related) and pragmatic (application oriented) structures in programs. To demonstrate this approach, the authors implemented a subset of this knowledge base, and a search mechanism called the Program Reference Language (PRL), which is able to (CONTINUED)

83 06 10 031

ITEM #20, CONTINUED: locate portions of programs based on a description provided by a user. This work is an applied research effort. It was motivated by issues discovered during a study of software maintenance problems in the Air Force, and is intended to be moved into application within seven years.

THE INTELLIGENT PROGRAM EDITOR

A Knowledge Based System for Supporting Program and Documentation Maintenance

Daniel G. Shapiro
Brian P. McCune

Advanced Information & Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, CA 94040

ABSTRACT

This paper presents work in progress towards a program development and maintenance aid called the Intelligent Program Editor (IPE), which applies artificial intelligence techniques to the task of manipulating and analyzing programs. The IPE is a knowledge based tool: it gains its power by explicitly representing textual, syntactic, and many of the semantic (meaning related) and pragmatic (application oriented) structures in programs. To demonstrate this approach, we implemented a subset of this knowledge base, and a search mechanism called the Program Reference Language (PRL), which is able to locate portions of programs based on a description provided by a user. This work is an applied research effort. It was motivated by issues discovered during a study of software maintenance problems in the Air Force, and is intended to be moved into application within 7 years.

1. INTRODUCTION

The effort and expense involved in software maintenance have been recognized as major limitations on the capabilities of current software systems. The difficulties arise for several reasons: first, although hardware costs have decreased, software expenses have skyrocketed owing to the higher cost of professional programmers. Second, as software projects have become more and more ambitious, the technical difficulty of making changes to the resulting programs has increased dramatically. As an illustration of this fact, the maintenance costs for large systems typically surpass the funds required for their initial development; as a case in point, the Defense Department now spends more than 3 billion dollars per year on software maintenance. These problems are addressed in part by the creation of standardized structured programming languages such as Ada, but in our opinion they will only be solved by the results of new research into automated programming support systems. We expect that many such tools will rely on the application of artificial intelligence techniques.

To gain better insight into the specific problems of software maintenance, AI&DS performed a study which analyzed software maintenance problems in the Air Force[1]. The study concluded that the process of comprehending the form and function of existing software (i.e., what it does and how it does it) is the most crucial step in the maintenance process. A number of tools which can affect that problem within the medium term (3 to 7 years) were defined.

This "comprehension problem" is revealed in many ways. To begin with, most programming installations have a high turnover rate of personnel and have trouble finding qualified replacements. As a result, the maintenance personnel are often unfamiliar with the program that is being maintained. At the same time, documentation is often unavailable, or of poor quality when it is available. This increases the difficulty of comprehending a given program. It is not easy to understand a program by directly reading the code because of the quantity of detail involved and also because coding standards are poorly enforced and rarely agreed upon. Finally, the process of isolating bugs, designing solutions, and determining the ramifications of changes is difficult in the presence of an incomplete understanding of the program's organization. The relative difficulty of this task is affected by the tools available to the programmer.

The software maintenance study identified a collection of tools designed to alleviate these problems, all of which rely on a sophisticated understanding of the structure of programs. In effect, they operate by transferring some of the expertise currently in the minds of programmers into a machine usable form that can be shared. Three of the most relevant tool ideas are summarized below.

The Intelligent Program Editor (IPE) is a knowledge-based tool for supporting the development and maintenance of software. It embodies a deep understanding of the structure of programs and of the manipulations which programmers typically apply to code. It can provide access to a variety of

This research was supported by the Air Force Office of Scientific Research under contract F49620-81-C-0067, the Office of Naval Research under contract N00014-82-C-0119, and Rome Air Development Center under contract F30602-80-C-0176.

intelligent tools, e.g., the Documentation Assistant described below.

The Documentation Assistant is a system that helps organize, obtain, maintain and access many different forms of documentation, ranging from line by line comments to design principles and application oriented requirements that underly the structure of code as a whole. The Documentation Assistant is intended to provide knowledge which other systems (such as the IPE) can employ.

The Programming Manager assists the programmer by systematically applying administrative and technical policies. It enforces some procedures (e.g., testing of code before installation), suggests others (e.g., notifying a user group of a change), and automatically performs some actions on its own. The Programming Manager is also intended as a form of Documentation Assistant for expressing heuristic knowledge about code, for example, that bugs in module A often cause run-time errors in module B.

At the current time, AI&DS is actively working on all of the tools described above. The remainder of this paper focuses on a description of the IPE and of the knowledge it will contain. We conclude with a scenario demonstrating an actual implementation of a portion of the IPE's knowledge base used in the context of a program search.

2. THE INTELLIGENT PROGRAM EDITOR

The Intelligent Program Editor (IPE) is a tool now being developed to support software development and maintenance in a sophisticated way[2]. The system gains its power through the use of an explicit model of the programming process, and a database called the Extended Program Model (the EPM), which represents the functional structure of code. The IPE uses this knowledge to support the design and manipulation of programs as semantic objects; this should be contrasted with the text string viewpoint that most editors provide. For example, the IPE will be able to automatically fill in syntactic forms, prompt the user during the completion of programming cliches, and monitor a program for semantic consistency while it is being modified. We expect the IPE to model the type of the user's programming activity, and to help choose or invoke appropriate tools.

The payoff of the IPE may be extremely large in terms of enhanced programmer productivity and increased reliability of code. Productivity will be improved because the system's high level vocabulary and manipulation methods will allow maintenance requests to be completed faster. Reliability will be enhanced because the IPE will automatically catch certain kinds of semantic errors that were formerly passed into delivered code. In addition, the IPE will have a large impact on the area of program comprehension; since it maintains a knowledge base that documents code from a variety of perspectives, it provides a forum for transferring expertise that was formerly lost as programmers moved on to different tasks and jobs. If these effects cumulatively produce as little as a one percent effect on the maintenance process in the U.S., the savings will be measured in the tens of millions of dollars annually.

Figure 1 contains a block diagram of the IPE. The system is composed of three major parts: the Extended Program Model, which provides knowledge of program structures and how to access them; a Programming Context Model, which lets the system understand some of the user's intent as he accesses or modifies code; and a collection of semantic analysis and manipulation tools that provide the programmer with a more powerful vocabulary for manipulating programs, above the level of character by character, or line by line changes. The IPE also contains a user interface and a programming executive which coordinate the facilities of the system and present them to the user. The user interface will use multiple windows and allow commands to be typed or selected from menus. See reference 2 for details.

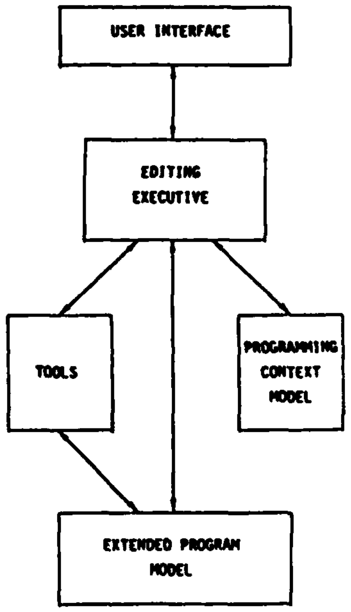


Figure 1: The Intelligent Program Editor (IPE)

2.1 THE PROGRAMMING CONTEXT MODEL

The Programming Context Model is a knowledge base that identifies the sequence of activities that are typically involved in the process of developing and maintaining code. This information supports the IPE in a variety of ways, but in particular it allows the system to guide the programmer through the coding sequence and to remind him of actions which he has not performed. For example, the Programming Context Model lists program creation, debugging, modification and exploration as major contexts, and refines program creation into functional definition, algorithm definition, data structure selection and coding. Since the coding process is further defined to include documentation, the IPE can invoke the Documentation Assistant tool and prompt the user to provide specific types of formatted information when each new module is defined.

The context model also gives the IPE a way to invoke its own facilities at appropriate times. For example, if the user is in the process of defining an algorithm, then the system will automatically search the EPM's database of typical programming patterns to see if a relevant template can be applied. (It should be mentioned that the IPE will not enforce a particular sequence of coding activities. Our philosophy is to allow the programmer to freely jump between programming contexts as he desires.)

2.2 MANIPULATION AND ANALYSIS TOOLS

The IPE's manipulation and analysis tools directly employ the knowledge sources in the EPM. These tools are responsible for making additions and deletions to the EPM's store of information, and for using its data to run semantic checks on the user's program as it is formed. (The EPM itself also does lower-level checking automatically to ensure the internal consistency and well-formedness of its multiple knowledge representations.) We have defined several tools of this kind that the IPE should contain. In addition to the Documentation Assistant discussed previously, the IPE will have an advanced program manipulation facility, a semantic error detection tool, and a style analysis capability. These are described in the following sections.

2.2.1 Advanced Program Manipulation

An intelligent editor that has a substantial amount of knowledge about the semantic structure of programs and about the semantics of meaningful operations can supply much better support to the programmer. For example, it is possible to provide operations that directly transform "while" loops into "for" loops, or iterations into recursions. Another type of syntactic operation interactively constructs a subroutine call by prompting the user for the name and actual parameters of a routine. This process will ensure that the number and type of the arguments in the calling statement agree with the declarations in the procedure's implementation.

The IPE will also provide templates for more semantic constructs, such as the typical programming patterns (described later) which are the building blocks that programmers use to implement larger algorithms or routines. With this information in hand, the system will be able to guide the user through the implementation of sophisticated routines by prompting for each functional part of a routine by using a mnemonic word or phrase.

2.2.2 Semantic Analysis

The semantic analysis tools within the IPE allow the system to identify sections of code which violate principles of correct program construction. These principles define "rational form" constraints which restrict the allowable composition of programs. For example, traditional type checking operations for strongly typed languages ensure that assignment statements are never used between variables that are declared to be of incompatible types. Similarly, it is not reasonable to use a variable before it is initialized. As a third example, a rational form constraint insists that all sections of a program can in fact be reached through some sequence of control steps (and yet, many large programs often contain dead regions which cannot be executed even in principle).

The semantic analysis tools perform these kinds of operations by examining the representations which the EPM provides. The data which supports these capabilities are described in Section 2.3.

2.2.3 Style Analysis

Some programming styles (patterns of programming language usage) are hard to comprehend and are subject to inadvertent or difficult-to-detect errors. Guidelines of good style include advice about making systems modular, adding comments to the code, clearly describing any assumptions made, and minimizing the use of "side-effects", etc.

Current automated style analysis tools are limited to straightforward syntactic analysis of code. Style analysis in the IPE will be similar in nature to the semantic analysis discussed above, except that the rules will be recommendations rather than requirements. By making the style analyzer a tool of the IPE, style analysis can be done on an incremental basis, e.g., each time a module is completed. The user can use all of the facilities of the IPE for altering code or documentation to conform to the style analysis guidelines. When appropriate, the IPE might be able to perform simple transformations to automatically correct style violations. In addition, the user would be provided with the ability to modify the style rules, so that ones which are not essential and which conflict with the user's preferred style can be suppressed.

In keeping with the philosophy of the IPE, style rules can be textual (e.g., "loop bodies should be indented"), syntactic (e.g., "don't assign to loop variables inside a loop"), semantic ("don't use expressions with side effects in declarations"), or even application oriented in nature.

2.3 THE EXTENDED PROGRAM MODEL

The Extended Program Model (EPM) is a system for representing and accessing programs in a sophisticated way. It accomplishes these tasks by defining a vocabulary for discussing programs which uses terms that are much closer to the ones which programmers naturally employ. The EPM provides this capability through the use of a knowledge base that represents the structure of programs from a variety of views: from low-level textual, or character by character information, to explicit semantic structures that document the programmer's intent for a piece of code. This information corresponds to what we believe the Documentation Assistant and the other manipulation and analysis tools discussed earlier need to use. Thus, the EPM can form the backbone for a number of systems which exhibit a deep understanding of the organizational structure and meaning of code.

The EPM is constructed in terms of two major subsystems (see Figure 2): a database of program structures (the PSDB) and a search and updating component called the Program Reference Language (or PRL), which provides access to the PSDB. In addition, the EPM contains a library of "rational form" constraints that will monitor program composition for its semantic content. As a whole, the system can be thought of as a database management system for maintaining code. It provides a search language for accessing its knowledge, a facility for performing updates, as well as a set of semantic integrity and consistency constraints for monitoring the validity of the data it contains.

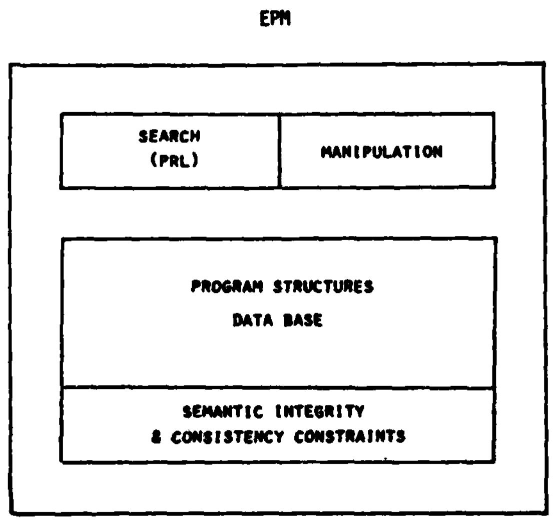

EPM
SEARCH (PRL)
MANIPULATION
PROGRAM STRUCTURES DATA BASE
SEMANTIC INTEGRITY & CONSISTENCY CONSTRAINTS

Figure 2. The Extended Program Model

2.3.1 The Program Structures Database

The EPM's knowledge or database of program structures (the PSDB) is constructed in terms of a hierarchy of representations which reflect the transition from a syntactic to a more intention-oriented analysis of code (Figure 3). For the purposes of the PRL, we are considering these viewpoints to be abstract data types which facilitate different sorts of retrieval operations.

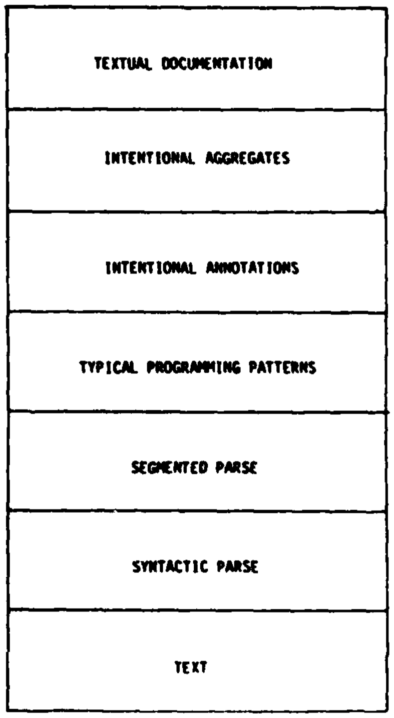


Figure 3. A Hierarchy of Program Structures in the EPM

The textual representation gives the EPM the view that most text editors provide. It is a low-level approach, concerned with words and delimiters as opposed to the semantics of programs, but it allows for important textual search operations. Similarly, the syntactic viewpoint is provided by some prototype text and "structure" editors. It embodies the rules of grammar for particular programming languages. The syntactic knowledge base provides the EPM with a vocabulary for programming constructs such as "for" loops, procedures, and the visible and private designations in Ada programs.

At the next level, we have provided a segmented parse abstraction which defines a vocabulary for a program in terms of its component data and control flow. So, for example, iterations are decomposed into a set of roles which identify the subfunctions of a loop. In the breakdown we are using, loops contain generators, filters, terminators, and augmentations[3]. Generators are segments which produce a sequence of values. They can be further refined into initializations and a body, which is the portion that is executed many times. Filters restrict that sequence of values. A terminator is like a filter, except that it has the additional potential to stop execution of the loop. An augmentation consumes values and produces results. There are other vocabulary elements for describing straight line code.

The taxonomy that has been discussed up until this point primarily captures information about the form of programs as opposed to their meaning. The only semantic elements we have introduced describe the substructure of built-in entities such as loops. In the next, more abstract viewpoint, we consider programs to be built of objects with stereotyped purposes. These are called typical programming patterns (TPPs). Examples of TPPs include variable interchanges, list insertions, and hash table abstractions. These abstractions are the tools employed by every expert programmer. Rich has defined a library of such TPPs[4].

The remaining knowledge bases, i.e., intentional annotations, intentional aggregates and textual documentation, all provide methods for associating the intentions behind a program with specific features of code. They often capture pragmatic knowledge relating to the domain of application of the program. For example, an intentional annotation might identify the author, creation date, and modification history of a particular file, or record comments about the goals and assumptions of a specific routine. Intentional aggregates associate larger program fragments with key words supplied by the programmer. They can be used to collect the TPPs and other program features that implement a single purpose.

The documentation knowledge base allows the user to associate textual comments with any of the program features already described. So, for example, he can document the data flow in a particular module (saying why an input-output relationship occurs), justify his use of particular TPPs, or explain why particular syntactic features are employed. This knowledge base takes advantage of the EPM's partitioning of program knowledge to classify comments in useful ways. For example, the textual documentation knowledge base is aimed at capturing some of the semantics implicitly associated with the textual comments that are normally attached directly to code.

3. THE PROGRAM REFERENCE LANGUAGE IMPLEMENTATION

In order to demonstrate the feasibility of the EPM, we implemented a portion of the knowledge base described above, and built a version of the EPM's search facility (the PRL) which operates on that data.

The PRL is a tool for locating regions of program text based upon a description provided by the user. As a support system, it provides programmers with a mechanism for isolating portions of programs in situations where they are not familiar with the detailed structure of the code. This occurs in the process of editing programs which are too large to remember explicitly, in the act of understanding code which has rarely been seen before, or in the process of completing partially implemented designs. In the context of program maintenance, the PRL helps to alleviate some of the burden on the programmer by supplying an intention-oriented vocabulary for referencing code.

The Program Reference Language Implementation (PRLI) allows program search based on four of the representations described above, namely the textual, syntactic, segmented parse and typical programming pattern views (Figure 4). These knowledge bases are connected through a "code region" abstraction that associates program features with physical sections of program text.

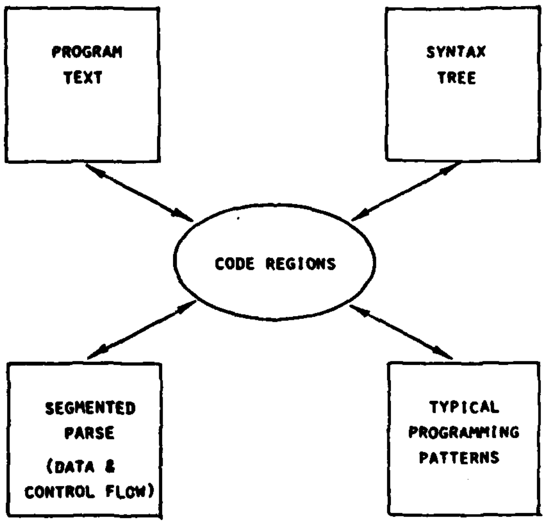

PROGRAM TEXT
SYNTAX TREE
CODE REGIONS
SEGMENTED PARSE (DATA & CONTROL FLOW)
TYPICAL PROGRAMMING PATTERNS

Figure 4. The Program Reference Language Implementation

The PRLI has a flat information structure. It represents each knowledge base in terms of a complex tree or graph structure of frames. However, the knowledge bases have no direct links between one another, although the system can arbitrarily convert between viewpoints by using code regions as an intermediary. These conversions are heuristic processes since the separate representations typically do not correspond on a one to one basis.

In the context of our applied work, we have also restricted the types of information the PRLI contains. The information in its database is either automatically available (based on current research prototypes), or can be reasonably obtained from the user. In situations where the latter is necessary, we assume that information may be provided in an incomplete form. It is important to note that every time a piece of documentation is added to the system's knowledge base, the performance of the PRL will increase. This should have the effect of encouraging the addition of information by the programmer, which has always been a major problem with the creation of documentation.

3.1 CODE PAINTING

From a computational point of view, the main problem involved with this multiple representation approach is to define a mechanism that is able to compare information obtained from the different sources of knowledge. The PRLI accomplishes this via the code region abstraction, which functions as a common language that each of the representations can use to "communicate".

Code regions support two different approaches to search. In the first method, which we call sequential filtering, the user makes a gross stab at selecting a code region by generating all of the elements which satisfy a fairly easy condition. He then sequentially restricts that set by applying more and more conditions. For example, to find "the loop which computes the sum of the test scores", he locates the set of all loops, and then restricts it to the ones which involve test scores and summations.

In the second approach, the user identifies a collection of items, possibly from several different knowledge bases, and intersects them together to find the elements which satisfy all of the conditions he wants to impose. In this "code painting" approach, the PRLI views each element of a knowledge base as a specification for a region of program text (meaning a portion of the program text); it combines them by essentially overlaying the corresponding regions of code. For example, the user locates the "loop which computes the sum of the test scores" by figuratively coloring all loops red and all places that compute the sums of test scores yellow. Any region which comes up orange has all of the properties that were desired. The implementation of code painting is described in reference 5.

Code painting is a deliberately coarse affair. It is designed to exploit the kind of incomplete or even slightly inaccurate information which the EPM will contain, given that much of the data is provided by the user. In many cases, code painting may not identify the exact section of the program which the user desired, but in the context of an interactive system with a screen oriented display, close will be good enough.

3.2 A SCENARIO USING THE PRLI

The following example shows how the PRLI uses the code painting paradigm to answer the question "find the initializations of the loop which computes the sum of the test scores", given the Ada program shown in Figure 5. (This is a modified version of an actual transcript that is presented in reference 5. A sequential filtering scenario is also provided there.)

In this example, the user starts by identifying three sets of data, corresponding to the summation TPPs, syntactic loops, and segmented parse frames involving the test score array.

```
> (index 'summation tpp-database)
   => TPPset1

> (index 'loops syntax-database)
   => LOOPset1:[length 2]

> (index 'TEST-SCORES segp-database)
   => SEGset1:[length 6]
```

The program only contains one TPP, but there are two loops, and several segments which relate to the variable TEST-SCORES. It is important to notice that these segments use the data contained in the variable TEST-SCORES but do not necessarily reference it by that name. For example, the literal "A(I)" in the ARRAYSUM function accesses the test score array. This correspondence is available from the data flow analysis within the segmented parse.

The user intersects these descriptions by invoking the code painting paradigm. The code-painting algorithm returns the largest region of text which can be described in all three ways.

```
for MAXSIZE in 1..10 loop
  TOTAL:= ARRAYSUM (TEST-SCORES, MAXSIZE);
  put (TOTAL);
end loop;


function ARRAYSUM (A: in ARRAY; N: in INTEGER) return INTEGER is
  begin
    SUM: REAL:= 0;
    for I in 1..N loop
      SUM:= SUM + A(I);
    end loop;
    return SUM;
end ARRAYSUM;
```

Figure 5. The Program Used in the Scenario

```
> (overlay-code-regions TPPset1 LOOPset1
   SEGset1)
   => CODE-REGION1
      **for I in 1..N loop
          SUM:= SUM + A(I);
        end loop;**
```

In order to compute this information, the *overlay* function automatically converts the input sets into their corresponding regions of code. In the case of the TPP, the programmer had to define that mapping at some time. The other translations are available, but heuristic in nature.

At this point, the user has identified a loop which computes the sum of the test scores. In order to find the initializations of this code, he views this region from the segmented parse perspective, and scans it for segments of the appropriate type. The term "initialization" is a segmented parse keyword.

```
> (Filter (Segs-Within CODE-REGION1)
   '(Seg-Type "initialization"))
     => SEGset2:[length 2]
```

The PMLI converts CODE-REGION1 to a set of segmented parse frames (a heuristic process), and the function Segs-Within enumerates the subsegments it contains. The system identifies two initializations as a result. The user prints them by converting them to the textual view.

```
> (show! SEGset2)
   => for I in **1..N** loop
   => **SUM: REAL := 0;**
```

The answers correspond to the initializations of the iteration variable "I", and the accumulation variable, "SUM". The PMLI retrieves the second initialization, even though it is lexically outside of the summation loop itself. It is identified from the segmented parse analysis, which associates a loop and its initializations no matter how far apart they might have been in the original code.

4. CURRENT STATUS AND FUTURE WORK

AI&DS is now developing a prototype version of the Intelligent Program Editor, which is intended to demonstrate the efficacy of our knowledge based approach to the design of programming support tools. The prototype will embody a portion of all of the facilities that have been described: the EPM, the PML, a collection of manipulation and analysis tools and the Program Context Model. The IPE is currently targeted for languages such as Ada and CMS-2. It will run initially on a Symbolics 3600, a fast, personal LISP computer that features a high-resolution bit-map display.

In terms of specific modifications, we expect to augment the EPM's knowledge base to include more pragmatic information (e.g., the relation between requirements and program structures), and we intend to extend the PML to the point where it will be able to automatically plan and carry out search requests of the kind demonstrated in this paper (based on a single user query). When these extensions are complete, the PML will define a more formal reference language.

The task of building a prototype for the IPE involves a number of issues including the incremental modification of knowledge bases, and the recognition of user intentions in code. In order to solve these problems in the context of our applied research, we expect to rely heavily on methods for eliciting information from the user, and to focus on template-oriented techniques for manipulating programs.

5. REFERENCES

1. Dean, Jeffrey S., and Brian P. McCune, "Advanced Tools for Software Maintenance", AI&DS TR 3006-1, October 1982.

2. Shapiro, Daniel G., Brian P. McCune, and Gerald A. Wilson, "Design of an Intelligent Program Editor", AI&DS TR 3023-1, September 1982.

3. Waters, Richard C., "Automatic Analysis of the Logical Structure of Programs", AI-TR-492, Artificial Intelligence Laboratory, MIT, 1978.

4. Rich, Charles, "Inspection Methods in Programming", AI-TR-604, Artificial Intelligence Laboratory, MIT, 1981.

5. Shapiro, Daniel G., and Brian P. McCune, "Searching a Knowledge Base of Programs and Documentation", AI&DS TM 1014-2, January 1983.

ATE

LME

8